Lecture 3:

Modules

Modules — An Overview

The `MODULE` program unit provides the following facilities:

- □ global object declaration;
- □ procedure declaration (includes operator definition);
- □ semantic extension;
- □ ability to control accessibility of above to different programs and program units;
- □ ability to package together whole sets of facilities;

Module - General Form

```
MODULE Nodule
 ! TYPE Definitions
 ! Global data
  ! ..
 ! etc ..
CONTAINS
  SUBROUTINE Sub(..)
   ! Executable stmts
  CONTAINS
    SUBROUTINE Int1(..)
    END SUBROUTINE Int1
     ! etc.
    SUBROUTINE Intn(..)
    END SUBROUTINE Int2n
  END SUBROUTINE Sub
   ! etc.
  FUNCTION Funky(..)
   ! Executable stmts
  CONTAINS
     ! etc
  END FUNCTION Funky
END MODULE Nodule
```

```
MODULE < module name >
        < declarations and specifications statements >
 [ CONTAINS
        < definitions of module procedures > ]
END [ MODULE [ < module name > ] ]
```

Modules — Global Data

Fortran 90 implements a new mechanism to implement global data:

- □ declare the required objects within a module;
- □ give them the `SAVE` attribute;
- □ `USE` the module when global data is needed.

For example, to declare `pi` as a global constant

```
MODULE  Pye
 REAL, SAVE :: pi = 3.142
END MODULE Pye

PROGRAM Area
 USE Pye
 IMPLICIT NONE
 REAL :: r
  READ*, r
  PRINT*, "Area= ",pi*r*r
END PROGRAM Area
```

`MODULE`s should be placed *before* the program.

Module Global Data Example

For example, the following defines a very simple 100 element integer stack

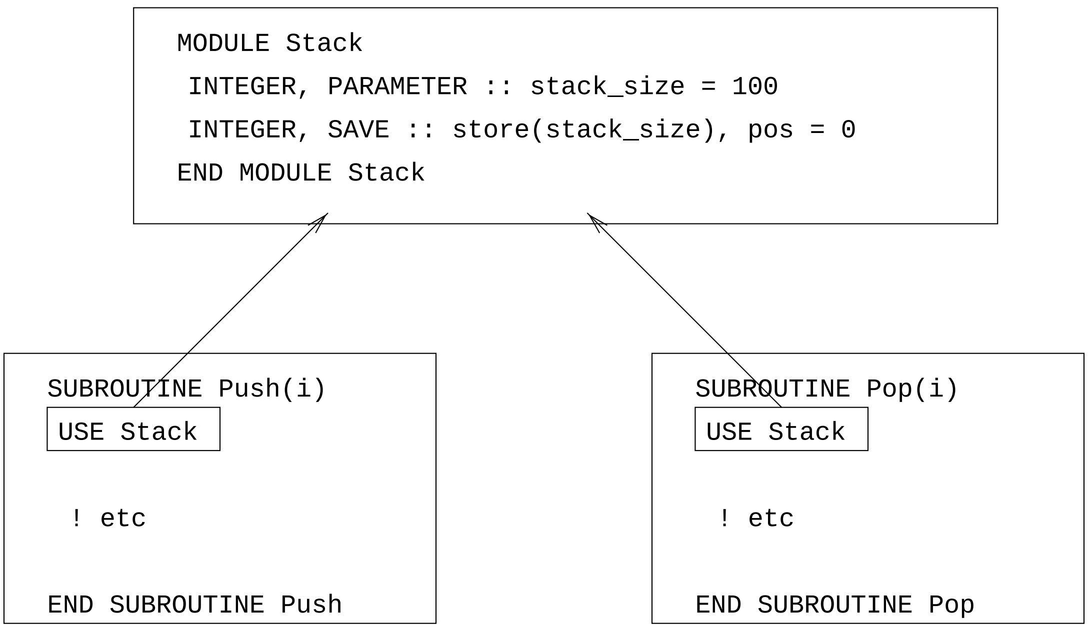
```
MODULE stack
 INTEGER, PARAMETER :: stack_size = 100
 INTEGER, SAVE :: store(stack_size), pos=0
END MODULE stack
```

and two access functions,

```
SUBROUTINE push(i)
 USE stack
 IMPLICIT NONE
  ...
END SUBROUTINE push
SUBROUTINE pop(i)
 USE stack
 IMPLICIT NONE
  ...
END SUBROUTINE pop
```

A main program can now call `push` and `pop` which simulate a 100 element `INTEGER` stack — this is much neater than using `COMMON` block.

Visualisation of Global Storage



Both procedures access the same (global) data in the `MODULE`.

Modules — Procedure Encapsulation

Module procedures are specified after the `CONTAINS` separator,

```
MODULE related_procedures
 IMPLICIT NONE
 ! INTERFACEs of MODULE PROCEDURES do
 ! not need to be specified they are
 ! 'already present'
CONTAINS
 SUBROUTINE sub1(A,B,C)
  ! Can see Sub2's INTERFACE
  ...
 END SUBROUTINE sub1
 SUBROUTINE sub2(time,dist)
  ! Can see Sub1's INTERFACE
  ...
 END SUBROUTINE sub2
END MODULE related_procedures
```

The main program attaches the procedures by *use-association*

```
PROGRAM use_of_module
 USE related_procedures ! includes INTERFACES
 CALL sub1((/1.0,3.14,0.57/),2,'Yobot')
 CALL sub2(t,d)
END PROGRAM use_of_module
```

`sub1` can call `sub2` or vice versa.

Encapsulation - Stack example

We can also encapsulate the `stack` program,

```
MODULE stack
 IMPLICIT NONE
 INTEGER, PARAMETER :: stack_size = 100
 INTEGER, SAVE :: store(stack_size), pos=0
CONTAINS
 SUBROUTINE push(i)
  INTEGER, INTENT(IN) :: i
   ...
 END SUBROUTINE push
 SUBROUTINE pop(i)
  INTEGER, INTENT(OUT) :: i
   ...
 END SUBROUTINE pop
END MODULE stack
```

Any program unit that includes the line:

```
USE stack
CALL push(2); CALL push(6); ..
CALL pop(i); ....
```

can access `pop` and `push` therefore use the 100 element global integer stack.

Modules — Object Based Programming

We can write a module that allows a derived type to behave in the same way as an intrinsic type. The module can contain:

- the type definitions,
- constructors,
- overloaded intrinsics,
- overload set of operators,
- other related procedures

An example of such a module is the varying string module which is to be an ancillary standard.

Derived Type Constructors

Derived types have in-built constructors, however, it is better to write a specific routine instead.

Purpose written constructors can support default values and will not change if the internal structure of the type is modified. It is also possible to hide the internal details of the type:

```
MODULE ThreeDee
 IMPLICIT NONE
 TYPE Coords_3D
  PRIVATE
  REAL :: x, y, z
 END TYPE Coords_3D
CONTAINS
 TYPE(Coords_3D) FUNCTION Init_Coords_3D(x,y,z)
  REAL, INTENT(IN), OPTIONAL :: x,y,z
  ! Set Defaults
  Init_Coords_3D = Coords_3D(0.0,0.0,0.0)
  IF (PRESENT(x)) Init_Coords_3D%x = x
  IF (PRESENT(y)) Init_Coords_3D%y = y
  IF (PRESENT(z)) Init_Coords_3D%z = z
 END FUNCTION Init_Coords_3D
END MODULE ThreeDee
```

If an argument is not supplied then the corresponding component of `Coords_3D` is set to zero.

Generic Interfaces

Most intrinsics are generic in that their type is determined by their argument(s). For example, the generic function `ABS(X)` comprises the specific functions:

- `CABS` — called when `X` is `COMPLEX`,
- `ABS` — called when `X` is `REAL`,
- `IABS` — called when `X` is `INTEGER`,

These specific functions are called the *overload set*.

A user may define his own overload set in an `INTERFACE` block:

```
INTERFACE CLEAR
 MODULE PROCEDUE clear_int
 MODULE PROCEDUE clear_real
END INTERFACE ! CLEAR
```

The *generic name*, `CLEAR`, is associated with *specific* names `clear_int` and `clear_real` (the overload set).

Generic Interfaces - Example

The full module would be

```
MODULE Schmodule
 IMPLICIT NONE
 INTERFACE CLEAR
  MODULE PROCEDURE clear_int
  MODULE PROCEDURE clear_real
 END INTERFACE CLEAR
CONTAINS
 SUBROUTINE clear_int(a)
  INTEGER, DIMENSION(:), INTENT(INOUT) :: a
   ... ! code to do clearing
 END SUBROUTINE clear_int
 SUBROUTINE clear_real(a)
  REAL, DIMENSION(:), INTENT(INOUT) :: a
   ... ! code to do clearing
 END SUBROUTINE clear_real
END MODULE Schmodule

PROGRAM Main
 IMPLICIT NONE
 USE Schmodule
 REAL :: prices(100)
 INTEGER :: counts(50)
  CALL CLEAR(prices) ! generic call
  CALL CLEAR(counts) ! generic call
END PROGRAM Main
```

The first procedure invocation would be resolved with `clear_real` and the second with `clear_int`.

Generic Interfaces - Commentry

In order for the compiler to be able to resolve the reference, both module procedures must be unique:

- the specific procedure to be used is determined by the *number, type, kind* or *rank* of the non-optional arguments,

- the overload set of procedures must be unambiguous with respect to their dummy arguments,

- default intrinsic types *should not* be used in generic interfaces, use parameterised types.

Basically, by examining the argument(s), the compiler calculates which specific procedure to invoke.

Overloading Intrinsic Procedures

When a new type is added, it is a simple process to add a new overload to any relevant intrinsic procedures.

The following extends the LEN_TRIM intrinsic to return the number of letters in the owners name for objects of type HOUSE,

```
MODULE new_house_defs
 IMPLICIT NONE
 TYPE HOUSE
  CHARACTER(LEN=16) :: owner
  INTEGER           :: residents
  REAL              :: value
 END TYPE HOUSE
 INTERFACE LEN_TRIM
  MODULE PROCEDURE owner_len_trim
 END INTERFACE
CONTAINS
 FUNCTION owner_len_trim(ho)
  TYPE(HOUSE), INTENT(IN) :: ho
  INTEGER :: owner_len_trim
  owner_len_trim = LEN_TRIM(ho%owner)
 END FUNCTION owner_len_trim
  .... ! other encapsulated stuff
END MODULE new_house_defs
```

The user defined procedures are added to the existing generic overload set.

Overloading Operators

Intrinsic operators, such as -, = and *, can be overloaded to apply to all types in a program:

- □ specify the generic operator symbol in an INTERFACE OPERATOR statement,

- □ specify the overload set in a generic interface,

- □ declare the MODULE PROCEDUREs (FUNCTIONs) which define how the operations are implemented.

These functions must have one or two non-optional arguments with INTENT(IN) which correspond to monadic and dyadic operators.

Overloads are resolved as normal.

Operator Overloading Example

The '*' operator can be extended to apply to the rational number data type as follows:

```
MODULE rational_arithmetic
 TYPE RATNUM
  INTEGER :: num, den
 END TYPE RATNUM
 INTERFACE OPERATOR (*)
  MODULE PROCEDURE rat_rat,int_rat,rat_int
 END INTERFACE
CONTAINS
  FUNCTION rat_rat(l,r)       ! rat * rat
   TYPE(RATNUM), INTENT(IN) :: l,r
    ...
     rat_rat = ...
  FUNCTION int_rat(l,r)       ! int * rat
   INTEGER, INTENT(IN)      :: l
   TYPE(RATNUM), INTENT(IN) :: r
   ...
  FUNCTION rat_int(l,r)       ! rat * int
   TYPE(RATNUM), INTENT(IN) :: l
   INTEGER, INTENT(IN)      :: r
   ...
END MODULE rational_arithmetic
```

The three new procedures are added to the operator overload set allowing them to be used as operators in a normal arithmetic expressions.

Example (Cont'd)

With,

```
USE rational_arithmetic
TYPE (RATNUM) :: ra, rb, rc
```

we could write,

```
rc = rat_rat(int_rat(2,ra),rb)
```

but better:

```
rc = 2*ra*rb
```

And even better still add visibility attributes to force user into good coding:

```
MODULE rational_arithmetic
 TYPE RATNUM
  PRIVATE
  INTEGER :: num, den
 END TYPE RATNUM
 INTERFACE OPERATOR (*)
  MODULE PROCEDURE rat_rat,int_rat,rat_int
 END INTERFACE
 PRIVATE :: rat_rat,int_rat,rat_int
  ....
```

Defining New Operators

can define new monadic and dyadic operators. They have the form,

$$.< name >.$$

Note:

- monadic operators have precedence over dyadic.
- names must be 31 letters (no numbers or underscore) or less.
- basic rules same as for overloading procedures.

Defined Operator Example

For example, consider the following definition of the .TWIDDLE. operator in both monadic and dyadic forms,

```
MODULE twiddle_op
 INTERFACE OPERATOR (.TWIDDLE.)
  MODULE PROCEDURE itwiddle, iitwiddle
 END INTERFACE ! (.TWIDDLE.)
CONTAINS
 FUNCTION itwiddle(i)
  INTEGER itwiddle
  INTEGER, INTENT(IN) :: i
  itwiddle = -i*i
 END FUNCTION
 FUNCTION iitwiddle(i,j)
  INTEGER iitwiddle
  INTEGER, INTENT(IN) :: i,j
  iitwiddle = -i*j
 END FUNCTION
END MODULE
```

The following

```
PROGRAM main
 USE twiddle_op
 print*, 2.TWIDDLE.5, .TWIDDLE.8, &
          .TWIDDLE.(2.TWIDDLE.5), &
          .TWIDDLE.2.TWIDDLE.5
END PROGRAM
```

produces

```
-10 -64 -100 20
```

Precedence

- user defined monadic operators are most tightly binding.
- user defined dyadic operators are least tightly binding.

For example,

```
.TWIDDLE.e**j/a.TWIDDLE.b+c.AND.d
```

is equivalent to

```
(((.TWIDDLE.e)**j)/a).TWIDDLE.((b+c).AND.d)
```

User-defined Assignment

Assignment between two different user defined types must be explicitly programmed; a `SUBROUTINE` with two arguments specifies what to do,

- the first argument is the result variable and must have `INTENT(OUT)`;
- the second is the expression whose value is converted and must have `INTENT(IN)`.

Overloading the assignment operator differs from other operators:

- assignment overload sets do **not** have to produce an unambiguous set of overloads;
- later overloads override earlier ones if there is an ambiguity;

Defined Assignment Example

Should put in a module,

```
INTERFACE ASSIGNMENT(=)
 MODULE PROCEDURE rat_ass_int, real_ass_rat
END INTERFACE
PRIVATE :: rat_ass_int, real_ass_rat
```

specify `SUBROUTINE`s in the `CONTAINS` block:

```
SUBROUTINE rat_ass_int(var, exp)
  TYPE (RATNUM), INTENT(OUT) :: var
  INTEGER, INTENT(IN) :: exp
  var%num = exp
  var%den = 1
END SUBROUTINE rat_ass_int
SUBROUTINE real_ass_rat(var, exp)
  REAL, INTENT(OUT) :: var
  TYPE (RATNUM), INTENT(IN) :: exp
  var = REAL(exp%num) / REAL(exp%den)
END SUBROUTINE real_ass_rat
```

Wherever the module is used the following is valid:

```
ra = 50
x = rb*rc
```

for real `x`.

Restricting Visibility

- Objects in a `MODULE` can be given visibility attributes:

```
PRIVATE :: rat_ass_int, real_ass_rat
PRIVATE :: rat_int, int_rat, rat_rat
PUBLIC :: OPRATOR(*)
PUBLIC :: ASSIGNMENT(=)
```

  only allows access to symbolic versions of multiply and assignment (* and =).

- This allows the internal structure of a module to be changed without modifying the users program.

- default visibility is `PUBLIC`, this can be reversed by a `PRIVATE` statement.

- individual declarations can also be attributed,

```
INTEGER, PRIVATE :: Intern
```

Derived Types with Private Components

The type `RATNUM` is declared with `PRIVATE` internal structure,

```
TYPE RATNUM
 PRIVATE
 INTEGER :: num, den
END TYPE RATNUM
```

The user is unable to access specific components,

```
TYPE (RATNUM) :: splodge
 splodge = RATNUM(2,3) ! invalid
 splodge%num = 2       ! invalid
 splodge%den = 3       ! invalid
 splodge = set_up_RATNUM(2,3) ! OK
! set_up_RATNUM must be module procedure
 CALL Print_out_RATNUM(splodge)
! Print_out_RATNUM must be module procedure
```

this allows the internal representation of the type to be changed:

```
TYPE RATNUM
 PRIVATE
 REAL :: numb
END TYPE RATNUM
```

Accessibility Example

We can update our stack example,

```
MODULE stack
 IMPLICIT NONE
 PRIVATE
 INTEGER, PARAMETER :: stack_size = 100
 INTEGER, SAVE :: store(stack_size), pos = 0
 PUBLIC push, pop
CONTAINS
 SUBROUTINE push(i)
  INTEGER, INTENT(IN) :: i
   ... ! as before
 END SUBROUTINE push
 SUBROUTINE pop(i)
  INTEGER, INTENT(OUT) :: i
   ... ! as before
 END SUBROUTINE pop
END MODULE stack
```

User cannot now alter the value of `store` or `pos`.

Another Accessibility Example

The visibility specifiers can be applied to all objects including type definitions, procedures and operators:

For example,

```
MODULE rational_arithmetic
 IMPLICIT NONE
 PUBLIC :: OPERATOR (*)
 PUBLIC :: ASSIGNMENT (=)
 TYPE RATNUM
  PRIVATE
  INTEGER :: num, den
 END TYPE RATNUM
 TYPE, PRIVATE :: INTERNAL
  INTEGER :: lhs, rhs
 END TYPE INTERNAL
 INTERFACE OPERATOR (*)
  MODULE PROCEDURE rat_rat,int_rat,rat_int
 END INTERFACE ! OPERATOR (*)
 PRIVATE rat_rat, int_rat, rat_int
   ... ! and so on
```

The type `INTERNAL` is only accessible from within the module.

The `USE` Renames Facility

The `USE` statement names a module whose public definitions are to be made accessible.

Syntax:

`USE` < *module-name* > `&`
        [,< *new-name* > `=>` < *use-name* >...]

module entities can be renamed,

```
    USE Stack, IntegerPop => Pop
```

The module object `Pop` is renamed to `IntegerPop` when used locally.

`USE ONLY` Statement

Another way to avoid name clashes is to only use those objects which are necessary. It has the following form:

`USE` < *module-name* > [`ONLY:`< *only*-list >...]

The < *only*-list > can also contain renames (**=>**).

For example,

```
USE Stack, ONLY:pos, &
        IntegerPop => Pop
```

Only `pos` and `Pop` are made accessible. `Pop` is renamed to `IntegerPop`.

The `ONLY` statement gives the compiler the option of including only those entities specifically named.

Semantic Extension Modules

The real power of the `MODULE` / `USE` facilities appears when coupled with derived types and operator and procedure overloading to provide *semantic extensions* to the language.

Semantic extension modules require:

- ☐ a mechanism for defining new types;
- ☐ a method for defining operations on those types;
- ☐ a method of overloading the operations so user can use them in a natural way;
- ☐ a way of encapsulating all these features in such a way that the user can access them as a combined set;
- ☐ details of underlying data representation in the implementation of the associated operations to be kept hidden (desirable).

This is an Object Oriented approach.